En este pequeño libro intentaremos dar una visión general de la *Programación Orientada al Objeto* (POO o también del inglés OOP = *Object Oriented Programming*).

Ya que la OOP no es un lenguaje de programación, puede aplicarse a cualquier lenguaje, y de hecho hoy en día está disponible en mayor o menor medida en todos los lenguajes tradicionales (C se ha convertido en C++, Pascal en Delphi, VB incorpora parte de la OOP) y no aparece un lenguaje nuevo sin que incluya OOP (como es el caso de Java). Es por esto que intentaremos que todo lo que aquí se diga pueda ser aplicado a cualquier lenguaje OOP.

El objetivo de este libro es, pues, el aprendizaje de los términos teóricos que todos manejamos hoy en día, en cualquier entorno de programación cuando nos referimos a la orientación a objetos.



Introducción a la OOP



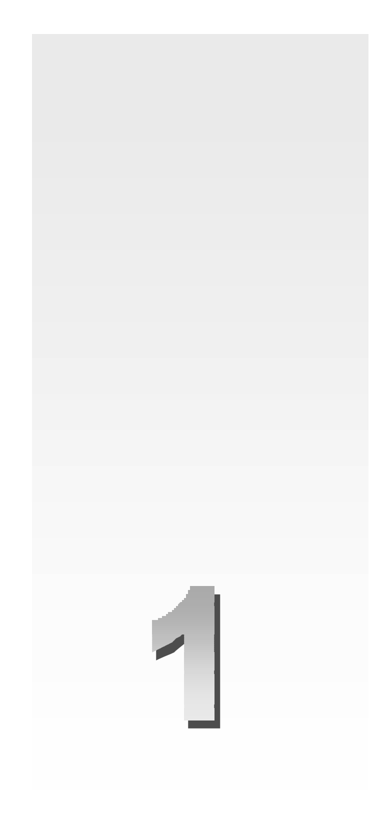
Grupo EIDOS

Introducción a la OOP. Versión 1.0.0

Autor: Francisco Morero

Ejemplar gratuito

Índice



Introducción

En este breve libro intentaremos dar una visión general de la ***Programación Orientada al Objeto*** (POO o también del inglés OOP = *Object Oriented Programming*). No hay acuerdo sobre cómo se debe traducir del inglés "Object Oriented Programming", si como "Programación Orientada a Objetos" o como "Programación Orientada al Objeto". Según me comentó un catedrático de lengua española, es más correcta la segunda acepción, y es por eso que es la que uso normalmente y la que se ha utilizado en este curso. El lector encontrará no obstante ambas acepciones dependiendo del autor y la editorial a se que se remita.

Ya que la OOP no es un lenguaje de programación, puede aplicarse a cualquier lenguaje, y de hecho hoy en día está disponible en mayor o menor medida en todos los lenguajes tradicionales (C se ha convertido en C++, Pascal en Delphi, VB incorpora parte de la OOP) y no aparece un lenguaje nuevo sin que incluya OOP (como es el caso de Java). Es por esto que intentaremos que todo lo que aquí se diga pueda ser aplicado a cualquier lenguaje OOP.

Sin embargo, cuando llegue el momento de escribir código, emplearemos una sintaxis híbrida de C++ y Java que según la opinión del autor del curso es la que puede resultar más clara independientemente del lenguaje al que se acaben aplicado estos conocimientos.

Para empezar, rogaríamos al lector experimentado en programación, que dejase a un lado todas sus ideas preconcebidas acerca de cómo debe realizarse un programa, la OOP tiene muy poco que ver con su forma habitual de trabajo, e incluso de ver el mundo; de hecho, los no programadores, captan mucho antes estas ideas, ya que no tienen prejuicios ni lastres ideológicos que les frenen.

Tenga en cuenta que usted, programador o no, ya sabe qué es un objeto y cómo manipularlo, lo lleva haciendo desde que nació; si entiende la similitud entre los objetos del mundo físico y los informáticos, la OOP le parecerá cosa de niños, pero no nos llevemos a engaño, éste es el punto más

escabroso, hacer ver qué es la OOP, quitarnos todas las ideas preconcebidas sobre programación tradicional.

Al autor, le costó más tiempo y esfuerzo llegar a verlo del que le hubiese costado a un neófito, pero yo llevo mucho tiempo programando y mis conceptos eran demasiado rígidos. Necesité por tanto una gran cantidad de "*dinamita intelectiva*" para hacer añicos la "*roca granítica*" de mis ideas preconcebidas.

Conscientes de ello, hemos considerado como la mejor aproximación exponer una serie de ejemplos y comparaciones de índole totalmente diferente. Algunas de ellas, las últimas, son para los programadores, el lector neófito, puede prescindir de estas, no son necesarias en absoluto para entender plenamente la OOP, pero a aquellos que conocen algún lenguaje de programación pueden serles útiles para entender esta técnica. Estos ejemplos los expondremos en el próximo capítulo, cuando entremos de lleno en OOP.

2

Breve revisión histórica

En este capítulo vamos a llevar a cabo una introducción muy general sobre la evolución de los lenguajes en informática, para entrar posteriormente y más a fondo, en describir la aparición, los principios teóricos y la evolución de la *Programación Orientada al Objeto*. Esta introducción no pretende ser un libro de referencia técnica de informática: hemos preferido dar una visión general y clara para el neófito, con lo que ello pueda acarrear de inexactitud, antes que perder la visión de conjunto.

Empezaremos por hacer una revisión de la evolución de los leguajes de programación, continuaremos explicando qué es la OOP para terminar haciendo una breve revisión histórica de la misma.

Evolución de los lenguajes de programación

Toda la historia de los lenguajes de programación se ha desarrollado en base a una sola idea conductora: hacer que la tarea de realizar programas para ordenadores sea cada vez lo más simple, flexible y portable posible.

La OOP supone, no solo un nuevo paso hacia ese fin, sino que además, a nuestro juicio, es el más importante acaecido hasta el momento. Como nos comenta Eckel: "*A medida que se van desarrollando los lenguajes, se va desarrollando también la posibilidad de resolver problemas cada vez más complejos. En la evolución de cada lenguaje, llega un momento en el que los programadores comienzan a tener dificultades a la hora de manejar programas que sean de un cierto tamaño y sofisticación.*" (Bruce Eckel, "Aplique C++", p. 5 Ed. McGrow-Hill).

Esta evolución en los lenguajes, ha venido impulsada por dos motores bien distintos:

Los avances tecnológicos

Los avances conceptuales (de planteamiento)

Los avances en cuanto a enfoque de la programación

Evolución en cuanto a la tecnología

Como usted probablemente sabrá, un ordenador no es más que un conjunto de microinterruptores, estos pueden estar apagados o encendidos (por lo que les llamamos elementos biestado). En un principio estos interruptores eran relés, que no es más que un electroimán que puede cerrar o abrir un circuito eléctrico. Si están apagados (no dejan pasar corriente), decimos que su estado es cero y en caso contrario uno.

Un programa de ordenador no es más que una sucesión de instrucciones que se ejecutarán secuencialmente, es decir, una detrás de otra. Por tanto, como la forma de programarlos es introduciendo secuencias de ceros y unos (lo que llamamos bits); puede imaginarse la gran probabilidad que existe de error al introducir sucesiones enormemente largas; además, una vez cometido un error, intentar encontrarlo y corregirlo puede llevarnos antes al manicomio con un grave cuadro de psicosis, que a la solución del problema.

De hecho, desde hace ya muchos años, los microprocesadores, que, como probablemente sabrá, son la maquinaria ejecutora de los ordenadores, se fabrican para ser programados, no en binario (secuencia de unos y ceros), sino en hexadecimal (un número hexadecimal equivale a 16 ceros o unos).

Cronológicamente el primer avance importante vino con la aparición de los **lenguajes ensambladores**. A estos lenguajes los llamamos de "bajo nivel", ya que se hallan estrechamente ligados a la forma de trabajo de la máquina con la que programamos y se opera con ellos de modo muy similar a como se hace cuando trabajamos en hexadecimal.

Quizás usted pueda pensar que no hay gran diferencia entre programar en hexadecimal (o en binario), y hacerlo a través de un ensamblador, y es cierto, pero este logro no lo es tanto en cuanto al modo de trabajo (que seguía siendo precario y tedioso) como en cuanto a la nueva concepción que ello implica: por primera vez aquello que nosotros escribimos no es entendible directamente por la máquina, sino que debe ser previamente traducido para que la máquina lo entienda.

Un lenguaje ensamblador lo único que hace es transcribir unos nemónicos (palabras fáciles de recordar) a la secuencia ceros y unos a los que el nemónico representa y que sí son entendibles por la máquina.

Por ejemplo, supongamos que deseamos almacenar un número en uno de los registros del microprocesador (un caso frecuente cuando se programa a bajo nivel). Supongamos que el número, para simplificar las cosas es el 4 (tiene la misma notación en decimal que en hexadecimal) y el registro el AX (uno de los registros de la familia de microprocesadores 80- de la marca Intel), la instrucción tendría un aspecto algo así:

HEXADECIMAL	ENSAMBLADOR
1A 01 04	MOV AX,04

Como podemos apreciar, es más fácil de recordar la segunda notación que la primera, haciéndose el código más inteligible.

El problema de los lenguajes ensambladores es que se hallan fuertemente comprometidos con la máquina. Un programa escrito en lenguaje ensamblador solo podrá ser ejecutado en la máquina donde fue diseñado.

El siguiente paso, vino con la aparición de los lenguajes de alto nivel.

El proceso de desarrollo de una aplicación con un lenguaje de alto nivel es mucho más rápido, simple y, por tanto, resulta más fácil detectar y corregir errores. Esto se debe principalmente a dos factores: por un lado, cada instrucción que escribimos en lenguaje de alto nivel puede equivaler a varias decenas e incluso cientos de instrucciones en ensamblador; por otra parte, la sintaxis de las instrucciones y los nemónicos que usamos se parecen algo más al lenguaje cotidiano (sobre todo si fuésemos angloparlantes).

Para hacernos una idea, diremos que realizar un programa que en un lenguaje de alto nivel nos puede llevar pocos días, en ensamblador es tarea de meses. Si tenemos esto en cuenta, comprenderemos que después de escribir un programa en código máquina y de haberlo depurado para eliminar los errores (*bugs* en terminología informática), a nadie le queden ganas de ponerse a re-escribirlo para otra máquina distinta. Con lo que la portabilidad queda mermada considerablemente. Hecho que no debe ocurrir con un lenguaje de alto nivel.

A estas alturas, es posible que usted esté pensado que todo esto es demasiado bonito para ser cierto y que en algún lugar debe estar el problema. Efectivamente, no nos llevemos a engaño, todo esto al final, se queda en papel mojado y buenas intenciones, debe ser que al dios de la computación le agrada hacernos la vida complicada.

La cuestión tiene dos vertientes bien distintas: por un lado, los programas realizados con lenguajes de alto nivel, realizan los mismo procesos de un modo más lento que de haberlos escrito con uno de bajo nivel, este problema no es demasiado gravoso si echamos un vistazo a la evolución en la potencia de cálculo de las máquinas en los últimos diez años. Salvo para programas muy específicos, o de gran rendimiento, no habría ningún problema, en escribir nuestras aplicaciones en un lenguaje de alto o bajo nivel.

¿Donde está el problema entonces? se preguntará usted. La respuesta es que está en la portabilidad.

En teoría, y solo en teoría, puesto que un lenguaje de alto nivel es independiente de la máquina donde se ejecute, el mismo código que escribimos para una máquina puede ser traducido al lenguaje hexadecimal en esa máquina o en otra cualquiera que disponga de traductor para ese lenguaje.

Por ejemplo, si yo escribo un programa en lenguaje BASIC para un ordenador, este mismo programa podría ejecutarse en cualquier otra máquina que dispusiese de traductor de BASIC para ella. Pero aquí aparece la torre de Babel de la informática. Existen traductores de lenguaje BASIC para multitud de máquinas diferentes, el problema reside en que cada máquina tiene un dialecto del lenguaje BASIC distinto a los demás, con lo que la portabilidad se hace imposible.

¿Y por qué ocurre esto? Eso ya son cuestiones de marketting, política de empresas y sociedad de consumo y libre mercado, algo que tiene muy poco que ver con los investigadores y las buenas intenciones.

Evolución en cuanto a la conceptualización

El primer avance en metodología de programación, vino con la **Programación Estructurada** (en este concepto vamos a incluir el propio y el de técnicas de **Programación con Funciones** -también llamado **procedural-**, ya que ambos se hallan íntimamente relacionados, no creemos, que se pueda concebir la programación estructurada sin el uso masivo de funciones).

La programación en ensamblador es lineal, es decir, las instrucciones se ejecutan en el mismo orden en que las escribimos. Podemos, sin embargo, alterar este orden haciendo saltos desde una instrucción a otro lugar del programa distinto a la instrucción que le sigue a la que se estaba procesando.

El BASIC tradicional también trabaja de este modo. Este sistema de trabajo es complicado, ya que obliga al programador a retener en su cabeza permanentemente todo el código escrito hasta un momento determinado para poder seguir escribiendo el programa; además a la hora de leerlo, el programador se pierde con facilidad porque debe ir saltando continuamente de unos trozos de código a otros.

Veamos un ejemplo típico de cómo se abordaría una misma tarea desde las dos perspectivas. La tarea consiste en mostrar los números del 1 a 10. Lo explicaremos en pseudo-código (exponer los pasos a realizar en lenguaje natural, en lugar de hacerlo en alguno de los lenguajes de programación existentes) para que resulte más comprensible:

Programación lineal

Cada línea de programa debe ir precedida de un identificador (una etiqueta) para poder referenciarla, para este ejemplo hemos utilizado números, aunque podría utilizarse cualquier otro identificador:

```
1. Hacer una variable igual a 0
2. Sumar 1 a esa variable
3. Mostrar la variable
4. Si la variable es 100 -> terminar, Si_no -> saltar a 1:
```

Programación estructurada

```
Hacer una variable igual a 0
Mientras que sea menor que 100 -> sumar 1 y mostrarla
```

Lo importante aquí, es que cuando escribimos un programa usando las técnicas de programación estructurada, los saltos están altamente desaconsejados, por no decir prohibidos; en cambio en BASIC, por ejemplo, son muy frecuentes (todos conocemos el prolífico **GOTO <nLínea>**), lo que no es nada conveniente si queremos entender algo que escribimos hace tres meses de forma rápida y clara.

De hecho, cuando el traductor (ya sea intérprete o compilador) cambia nuestro programa a código máquina, lo convierte a estilo lineal, pero eso es asunto de la máquina, nosotros escribimos y corregimos nuestro programa de un modo claro, y podemos seguir el flujo de la información con facilidad.

Lo que se intenta, es de que el programador pueda hacer programas cada vez más extensos sin perderse en un entramado de líneas de código interdependientes. Un programa en estilo lineal se parece a la red neuronal del cerebro, si usted ha visto alguna vez una foto de estas, no lo habrá

olvidado: en esa maraña de neuronas y conexiones nerviosas no hay quien se entienda, y si no que se lo pregunten a los neurofisiólogos.

Para evitar esto, junto con la programación estructurada aparece un concepto que nos permite abarcar programas más amplios con menor esfuerzo: el de **función**.

La idea es muy simple: muchas veces realizo procesos que se repiten y en los que sólo cambia algún factor, si trato ese proceso como un subprograma al que llamo cada vez que lo necesito, y cada vez que lo llamo puedo cambiar ese factor, estaré reduciendo el margen de error, al reducir el número de líneas que necesito en mi programa, ya que no tengo que repetir todas esas líneas cada vez que quiera realizar el proceso, con una sola línea de llamada al subprograma será suficiente; además, de haber algún fallo en este proceso el error queda circunscrito al trozo de código de la función.

Así, las funciones podemos entenderlas como unas cajas negras, que reciben y devuelven valores. Solo tengo que programarlas una vez, las puedo probar por separado y comprobar que funcionan correctamente, una vez terminadas puedo olvidarme de cómo las hice y usarlas siempre que quiera.

Simultáneamente al concepto de función aparece el de **variables de ámbito reducido**. Todos recordaremos nuestros nefastos días del BASIC, en los que cualquier variable usada en el programa, era conocida en todo el programa. Resultado: cuando habíamos definido 500 variables, no nos acordábamos para qué servía cada una ni que nombres habíamos usado y cuales no. Con un lenguaje estructurado, las variables son conocidas solo por aquellas partes del programa donde nos interesa que lo sean; pudiendo re-usar los nombres de las variables sin que haya colisión, siempre que estas se utilicen en ámbitos distintos.

Programación Orientada al Objeto

Por último, llegamos al más reciente avance, la OOP, que nos ofrece mucho mayor dominio sobre el programa liberándonos aún más de su control. Hasta ahora, el control del programa era tarea del programador, si usted ha realizado algún programa de magnitud considerable lo habrá padecido. El programador tenía que controlar y mantener en su mente cada proceso que se realizaba y los efectos colaterales que pudieran surgir entre distintos procesos, lo que llamamos colisiones. En OOP, el programa se controla a sí mismo y la mente del programador se libera enormemente pudiendo realizar aplicaciones mucho más complejas al exigir menor esfuerzo de atención, ya que los objetos son entidades autónomas que se controlan (si han sido creados correctamente) a sí mismos. Esto es posible principalmente porque los objetos nos impiden mezclar sus datos con otros métodos distintos a los suyos.

En programación estructurada, una función trabaja sobre unos datos, y no debería modificar datos que no le corresponde hacer, pero de eso tiene que encargarse el programador, en OOP es el propio sistema de trabajo el que impide que esto ocurra. Además, la re-usabilidad del código escrito es mucho mayor que con el uso de funciones, y las portabilidad es también mayor.

Pero este es el tema del que hablaremos en el resto del libro.

Evolución en cuanto al enfoque

La evolución de los lenguajes de programación, en cuanto a enfoque es también una evolución conceptual, pero ésta es tan profunda que supone un cambio drástico en cuanto al modo de concebir el tratamiento de la programación.

En este sentido, y dependiendo del autor a quien se consulte, existen dos o tres enfoques diferentes:

Programación procedural

Programación declarativa

Programación orientada a objetos

Programación Procedural

Casi todos los lenguajes que conocemos trabajan de forma procedural. Java, C, Pascal, BASIC, Cobol, Fortran, APL, RPG, Clipper, etc.

En ellos, debemos establecer, hechos (datos), reglas para el manejo de esos datos y de decisión y tenemos que decirle al lenguaje cómo alcanzar el objetivo que se persigue. Es decir, donde buscar la información, cómo manipularla, cuando parar, etc.

Si usted es programador de algún lenguaje procedural, todo esto le parecerá obvio, pero no funciona así con los lenguajes declarativos.

No vamos a ahondar más en estos lenguajes, porque suponemos que usted conoce algo de ellos.

Programación Declarativa

Los lenguajes más conocidos que existen hasta ahora, salvo **PROLOG**, son todos procedurales, éste es declarativo.

ProLog es acrónimo de PROgramming in LOGic. Este lenguaje fue desarrollado en la universidad de Marsella hacia 1970 por Alan Clomerauer y sus colegas.

ProLog, se basa en manipulaciones lógicas, utiliza la lógica proposicional -lógica de predicados- para realizar sus deducciones.

En ProLog no programamos, sino que declaramos hechos, es la maquinaria del lenguaje quien se encargará de extraer las conclusiones que resulten inferibles de estos hechos.

A esta maquinaria se le llama motor de inferencias, que es, por otro lado, el corazón de un Sistema Experto. Probablemente de este tipo de programas -los más famosos de la Inteligencia Artificial-, habrá usted oído hablar.

Programación Orientada al Objeto

Nosotros disentimos parcialmente de autores como Manuel Fonseca y Alfonso Alcalá, que incluyen la OOP como un nuevo enfoque. Ya que, entre otras cosas y principalmente, la OOP es un conjunto de técnicas.

Aclaremos la diferencia entre técnica y lenguaje de programación:

Una técnica de programación no es, obviamente, un lenguaje, pero puede aplicarse a cualquier lenguaje. Podemos definirlo como un conjunto de reglas a seguir para hacernos la tarea de programar más fácil. Son consejos de expertos programadores, que tras años de trabajo, han acumulado una gran experiencia. Pero estas técnicas, son obviamente, independientes del lenguaje en que trabajemos.

Un lenguaje de programación, es algo que todos más o menos conocemos: un conjunto de instrucciones entendibles directamente o traducibles al lenguaje del ordenador con el que trabajemos; combinando estas instrucciones realizamos programas.

Es cierto sin embrago, que para poder aplicar OOP al 100%, es necesario que el lenguaje nos proporcione una serie de mecanismos inherentes al propio lenguaje.

En cualquier caso, la OOP es casi 100% procedural y, desde luego, no es en absoluto declarativa.

De todos modos, el autor de este libro comparte en cierto modo la teoría de los tres enfoques. El problema, es el mismo que aparece a la hora de dilucidar donde acaba un color y comienza otro en el arco iris. Siempre que hacemos una clasificación de la realidad, nos encontramos con elementos que son fronterizos entre una clase y otra y son de difícil ubicación. Sin embargo, no queda más remedio, a efectos didácticos que realizarlas. Nos sentimos más inclinados limitar los enfoques en programación a los dos primeros, dejando fuera a la OOP.

Posiblemente, y con el tiempo, reconsideremos esta postura y estemos de acuerdo con estos y otros autores; al fin y al cabo, la OOP aún no está completamente asentada, y no sabemos donde nos habrá llevado en los próximos años.

¿Qué es la OOP?

Comenzaremos dando una definición por exclusión de lo que es la OOP, es decir, vamos a decir primero qué no es la OOP, para, luego, intentar dar una definición de lo que es.

LA OOP **no** es:

Un sistema de comunicación con los programas basados en ratones, ventanas, iconos, etc. Puesto que, normalmente, los lenguajes de OOP suelen presentar estas características y puesto que habitualmente estos entornos suelen desarrollarse con técnicas de OOP, algunas personas tiende a identificar OOP y entornos de este tipo. De igual modo a como se tendía a identificar la inteligencia artificial con lenguajes como LISP o PROLOG. El autor de este libro asegura haber visto más de un programa escrito en estos lenguajes que no tenía nada de inteligente y mucho menos de inteligencia artificial.

No es un lenguaje. De hecho las técnicas de OOP pueden utilizarse en cualquier lenguaje conocido y los que están por venir, aunque estos últimos, al menos en los próximos años, incluirán facilidades para el manejo de objetos. Desde luego, que en los lenguajes que prevén el uso de objetos la implementación de las técnicas de OOP resulta mucho más fácil y provechosa que los otros. Pero del mismo modo a lo comentado en el punto anterior, se pueden utilizar estos lenguajes sin que los programas resultantes tengan nada que ver con la OOP.

Como ya hemos comentado, la OOP son un conjunto de técnicas que nos permiten incrementar enormemente nuestro proceso de producción de software; aumentando drásticamente nuestra productividad por un lado y permitiéndonos abordar proyectos de mucha mayor envergadura por oto.

Usando estas técnicas, nos aseguramos la re-usabilidad de nuestro código, es decir, los objetos que hoy escribimos, si están bien escritos, nos servirán para "siempre".

Hasta aquí, no hay ninguna diferencia con las funciones, una vez escritas, estas nos sirven siempre. Pero es que, y esto sí que es innovador, con OOP podemos re-usar ciertos comportamientos de un objeto, ocultando aquellos otros que no nos sirven, o redefinirlos para que los objetos se comporten de acuerdo a las nuevas necesidades.

Veamos un ejemplo simple: si tenemos un coche y queremos que sea más rápido, no construimos un coche nuevo; simplemente le cambiamos el carburador por otro más potente, cambiamos las ruedas por otras más anchas para conseguir mayor estabilidad y le añadimos un sistema turbo. Pero seguimos usando todas las otras piezas de nuestro coche.

Desde el punto de vista de la OOP ¿Qué hemos hecho?

Hemos modificado dos cualidades de nuestro objeto (métodos): el carburador y las ruedas.

Hemos añadido un método nuevo: el sistema turbo.

En programación tradicional, nos hubiésemos visto obligados a construir un coche nuevo por completo.

Dicho en términos de OOP, si queremos construir un objeto que comparte ciertas cualidades con otro que ya tenemos creado, no tenemos que volver a crearlo desde el principio; simplemente, decimos qué queremos usar del antiguo en el nuevo y qué nuevas características tiene nuestro nuevo objeto.

Aún hay más, con OOP podemos incorporar objetos que otros programadores han construido en nuestros programas, de igual modo a como vamos a una tienda de bricolaje y compramos piezas de madera para ensamblarlas y montar una estantería o una mesa. Pero, es que además podemos modificar los comportamientos de los objetos construidos por otros programadores *sin tener que saber cómo los han construido ellos.*

Como puede ver, esto supone realmente una nueva concepción en el desarrollo de programas, algo radicalmente nuevo y de una potencia y versatilidad hasta ahora inimaginables. Si usted es programador de la "vieja escuela" le parecerá increíble, sin embargo, y como podrá comprobar, es totalmente cierto.

Breve historia de la OOP

Los conceptos de *clase* y *herencia* fueron implementados por vez primera en el lenguaje **Simula 67** (el cual no es sino una extensión de otro más antiguo, llamado Algol 60), este fue diseñado en 1967 por Ole-Johan Dhal y Krysten Nygaard en la Universidad de Oslo y el Centro de Computación Noruego (Norsk Regnesentral).

La historia de Simula, que es como se le llama coloquialmente, es tan frecuente como desafortunada. Fue diseñado como un lenguaje de propósito general y pasó por el mundo de la informática sin pena ni gloria durante años. Fue mucho después, con la aparición de otros lenguajes que se basaban en estos innovadores conceptos (Smalltalk y sobretodo C++), cuando se le reconoció a los creadores de Simula su gran mérito. Sin embargo, Simula sigue sin usarse porque estos conceptos han sido ampliados y han aparecido otros nuevos que le dan mayor potencia y flexibilidad a los conceptos originales de clase y herencia, conformando lo que hoy entendemos por *Programación Orientada al Objeto*.

Aunque Simula fue el padre de todo este revuelo, ha sido **Smalltalk** quién dio el paso definitivo y es éste el que debemos considerar como el primer lenguaje de programación orientado a objetos. Smalltalk fue diseñado (cómo no) en el *Palo Alto Research Center* (PARC) de Xerox Corporation's, en California.

Este ha sido uno de los centros de investigación que más avances ha dado al la informática en toda su historia; fue aquí donde se desarrolló el entorno de ventanas que hoy usan Windows en MS-DOS y XWindows en UNIX, los famosos ratones como dispositivos de entrada de datos o interfaces de usuario como el DataGlobe. Según últimas noticias, ahora andan desarrollando unos nuevos conceptos

de sistemas operativos con imágenes tridimensionales en movimiento que serán los que probablemente utilizaremos dentro de algunos años.

En este centro de investigación de Palo Alto, a comienzos de los 70, el proyecto iniciado por Alan Kay vio la luz con el nombre de Smalltalk. Lo que había empezado como un proyecto de desarrollo de un lenguaje de propósito general acabó siendo mucho más que eso, convirtiéndose en el origen de la, hasta ahora, última y más importante revolución en el desarrollo de software.

Smalltalk incluye no solo un lenguaje para el desarrollo de aplicaciones, sino que además incorpora herramientas de ayuda al desarrollo (p.ej. manejadores de árboles de clases, examinadores de objetos, etc.) y un completo interfaz gráfico de usuario.

El último gran paso, a nuestro juicio, lo dio Bjarne Stroustrup con la creación del **C++**, quizás el lenguaje de programación orientado a objetos más usado actualmente. Este, fue definido en 1986 por su autor en un libro llamado *The C++ Programming Language*, de cita y referencia obligadas cuando se habla de OOP. Tan importante es esta publicación, que cuando se habla de C++, a este libro se le llama "El Libro". Cuando algún experto se encuentra con alguna duda sobre cómo debería ser un lenguaje orientado al objeto recurre a él, y si no encuentra solución, se dirige directamente a Stroustrup.

La importancia del C++ radica, en que, abandonando ciertos requerimientos de los lenguajes de cuarta generación con tecnología OOP como son Smalltalk o Actor, ha conseguido darle una gran potencia y flexibilidad al más famoso lenguaje, el C.

Llegados a este punto se hace necesario aclarar que los lenguajes de OPP, podemos clasificarlos en **puros** e **híbridos**. Diremos que un lenguaje es OOP puro, cuando se ajusta completamente a los principios que esta técnica propone y contempla la posibilidad de trabajar exclusivamente con clases. Diremos que un lenguaje es híbrido de OOP y algún otro, cuando ese lenguaje, que normalmente existía antes de la aparición de la OOP, incorpora en mayor o menor medida facilidades para trabajar con clases.

De este modo, C++ es un lenguaje OOP híbrido. De hecho, C++ no incorpora todas las características de un lenguaje OOP, y no lo hace principalmente, porque es un lenguaje compilado y ello impide que se resuelvan ciertas referencias en tiempo de compilación necesarias para dotar a las clases de algunas de sus cualidades puramente OOP (a menos que se le dote de un motor OOP interno, el cual tiene en parte, pero esto es harina de otro costal y no forma parte de la finalidad de este libro).

Hoy en día, casi todos los lenguajes más usados en los 80 se han reconvertido en mayor o menor medida a la OOP, y han aparecido otros nuevos. Veámoslos rápidamente:

C ha pasado a llamarse **C++**. Es la mejor implementación OOP sobre un lenguaje compilado existente en este momento. Su punto más conflictivo no obstante no su total implementación OOP, sino el hecho de permitir herencia múltiple, lo que, según los teóricos, no es muy aconsejable.

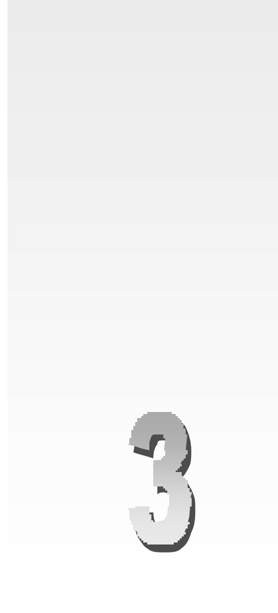
Pascal ha sido reciclado por Borland, pasando a llamarse **Delphi**. Tiene una implementación OOP bastante buena, pero al no permitir sobrecarga de funciones pierde una de las características de OOP: *Polimorfismo*.

Basic ha pasado a llamarse **Visual Basic** en manos de Microsoft. Este, aun siendo uno de los lenguajes más famosos del momento, no es un lenguaje OOP completo, ya que no incluye la característica más importante de la OOP: la *Herencia*.

Java es la estrella de los últimos años: es pura OOP, impecablemente concebido e implementado por Sun, y al que sólo se le puede achacar (si es que nos ponemos quisquillosos) que no dispone de sobrecarga de operadores, cualidad que de los citados aquí sólo dispone C++.

Incluso el viejo **Cobol** se ha reciclado, existiendo en estos momentos más de una versión de Cobol que incluye OOP.

Incluso el viejo **Cobol** se ha reciclado, existiendo en estos momentos más de una versión de Cobol que incluye OOP.



Programación Orientada al Objeto

Lo que pretendemos con este capítulo es que usted entienda los conceptos que aquí se le van a exponer, no importa que no entienda cómo llevarlos a la práctica, o dicho de un modo más técnico, cómo implementarlos en un lenguaje concreto (C, Java, Visual Basic, FoxPro, etc.). De esto ya nos encargaremos más adelante.

El concepto de *Sistema de Programación Orientado al Objeto* -Object Oriented Programming System (OOPS)-, y que nosotros llamamos OOP, agrupa un conjunto de técnicas que nos permiten desarrollar y mantener mucho más fácilmente programas de una gran complejidad.

Veamos ahora cuales son los conceptos relacionados con la OOP.

Conceptos básicos

En este capítulo veremos cuales son los principales conceptos relacionados con la OOP.

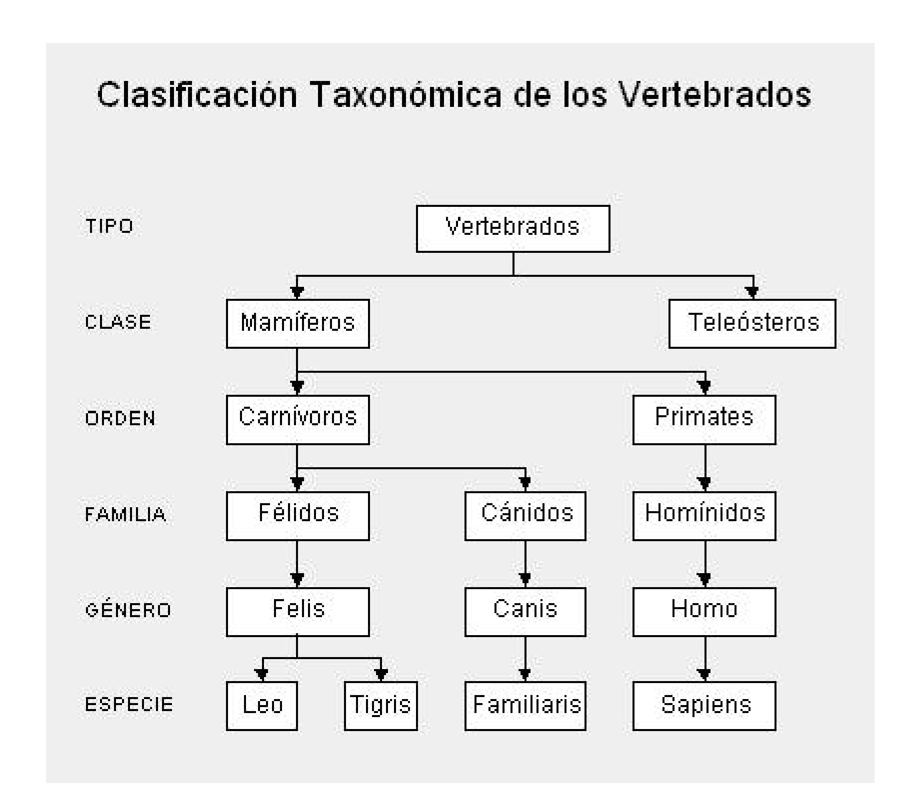
Estudiaremos los conceptos de *Clase, Objeto, Herencia, Encapsulación* y *Polimorfismo*. Estas son las ideas más básicas que todo aquel que trabaja en OOP debe comprender y manejar constantemente; es por lo tanto de suma importancia que los entienda claramente. De todos modos, no se preocupe si al finalizar el presente capítulo, no tiene una idea clara de todos o algunos de ellos: con el tiempo los irá asentando y se ira familiarizando con ellos, especialmente cuando comience a trabajar creando sus propias clases y jerarquías de clases.

Definición de Clase

Una *clase*, es simplemente una abstracción que hacemos de nuestra experiencia sensible. El ser humano tiende a agrupar seres o cosas -objetos- con características similares en grupos -clases-. Así, aun cuando existen por ejemplo multitud de vasos diferentes, podemos reconocer un vaso en cuanto lo vemos, incluso aun cuando ese modelo concreto de vaso no lo hayamos visto nunca. El concepto de *vaso* es una abstracción de nuestra experiencia sensible.

Quizás el ejemplo más claro para exponer esto lo tengamos en las taxonomías; los biólogos han dividido a todo ser (vivo o inerte) sobre la tierra en distintas clases.

Tomemos como ejemplo una pequeña porción del inmenso árbol taxonómico:



Ellos, llaman a cada una de estas parcelas *reino, tipo, clase, especie, orden, familia, género,* etc.; sin embargo, nosotros a todas las llamaremos del mismo modo: *clase*. Así, hablaremos de la clase animal, clase vegetal y clase mineral, o de la clase félidos y de las clases leo (león) y tigris (tigre).

Cada clase posee unas cualidades que la diferencian de las otras. Así, por ejemplo, los vegetales se diferencian de los minerales -entre otras muchas cosas- en que los primeros son seres vivos y los minerales no. De los animales se diferencian en que las plantas son capaces de sintetizar clorofila a partir de la luz solar y los animales no.

Como vemos, el ser humano tiende, de un modo natural a clasificar los objetos del mundo que le rodean en clases; son definiciones estructuralistas de la naturaleza al estilo de la escuela francesa de Saussure.

Prosigamos con nuestro ejemplo taxonómico y bajemos un poco en este árbol de clases.

Situémonos en la clase *felinos* (felis), aquí tenemos varias subclases (géneros en palabras de los biólogos): león, tigre, pantera, gato, etc. cada una de estas subclases, tienen características comunes (por ello los identificamos a todos ellos como felinos) y características diferenciadoras (por ello distinguimos a un león de una pantera), sin embargo, ni el león ni la pantera en abstracto existen, existen leones y panteras particulares, pero hemos realizado una abstracción de esos rasgos comunes a todos los elementos de una clase, para llegar al concepto de león, o de pantera, o de felino.

La *clase león* se diferencia de la *clase pantera* en el color de la piel, y comparte ciertos atributos con el resto de los felinos -uñas retráctiles por ejemplo- que lo diferencian del resto de los animales. Pero la clase león, también hereda de las clases superiores ciertas cualidades: columna vertebral (de la clase vertebrados) y es alimentado en su infancia por leche materna (de la clase mamíferos).

Vemos cómo las clases superiores son más generales que las inferiores y cómo, al ir bajando por este árbol, vamos definiendo cada vez más (dotando de más cualidades) a las nuevas clases.

Hay cualidades que ciertas clases comparten con otras, pero no son exactamente iguales en las dos clases. Por ejemplo, la clase hombre, también deriva de la clase vertebrado, por lo que ambos poseen columna vertebral, sin embrago, mientras que en la clase hombre se halla en posición vertical, en la clase león la columna vertebral está en posición horizontal.

En OOP existe otro concepto muy importante asociado al de clase, el de "*clase abstracta*". Una clase abstracta es aquella que construimos para derivar de ella otras clases, pero de la que no se puede instanciar. Por ejemplo, la clase mamífero, no existe como tal en la naturaleza, no existe ningún ser que sea tan solo mamífero (no hay ninguna instanciación directa de esa clase), existen humanos, gatos, conejos, etc. Todos ellos son mamíferos, pero no existe un animal que sea solo mamífero.

Del mismo modo, la clase que se halla al inicio de la jerarquía de clases, normalmente es creada sólo para que contenga aquellos datos y métodos comunes a todas las clases que de ella derivan: Son clases abstractas. En árboles complejos de jerarquías de clases, suele haber más de una clase abstracta.

Este es un concepto muy importante: el de "*clase abstracta*". Como hemos dicho, una *clase abstracta* es aquella que construimos para derivar de ella otras clases, pero de la que no se puede instanciar. Por ejemplo, la clase mamífero, no existe como tal en la naturaleza, no existe ningún ser que sea tan solo mamífero (no hay ninguna instanciación directa de esa clase), existen humanos, gatos, conejos, etc. Todos ellos son mamíferos, pero no existe un animal que sea solo mamífero.

Por último, adelantemos algo sobre el concepto de objeto.

El león, como hemos apuntado antes, no existe, igual que no existe el hombre; existen leones en los circos, en los zoológicos y, según tenemos entendido, aún queda alguno en la sabana africana. También existen hombres, como usted, que está leyendo este libro (hombre en un sentido neutro, ya sea de la subclase mujer o varón), o cada uno de los que nos encontramos a diario en todas partes.

Todos estos hombres comparten las características de la clase hombre, pero son diferentes entre sí, en estatura, pigmentación de la piel, color de ojos, complexión, etc. A cada uno de los hombres particulares los llamamos "objetos de la clase hombre". Decimos que son objetos de tipo hombre o que pertenecen a la clase hombre. Más técnicamente, *José Luis Aranguren* o *Leonardo da Vinci* son instanciaciones de la clase hombre; instanciar un objeto de una clase es crear un nuevo elemento de esa clase, cada niño que nace es una nueva instanciación a la clase hombre.

Definición de Objeto

Según el Diccionario del Uso del Español de María Moliner (Ed. Gredos, 1983), en la tercera acepción del termino *objeto* podemos leer: "Con respecto a una acción, una operación mental, un sentimiento, etc., cosa de cualquier clase, material o espiritual, corpórea o incorpórea, real o imaginaria, abstracta o concreta, a la cual se dirigen o sobre la que se ejercen."

No se asuste, nuestra definición de objeto, como podrá comprobar es mucho más fácil. En OOP, un objeto es un **conjunto de datos y métodos**; como imaginamos que se habrá quedado igual, le vamos a dar más pistas.

Los datos son lo que antes hemos llamado características o atributos, los métodos son los comportamientos que pueden realizar.

Lo importante de un sistema OOP es que ambos, datos y métodos están tan intrínsecamente ligados, que forman una misma unidad conceptual y operacional. En OOP, no se pueden desligar los datos de los métodos de un objeto. Así es como ocurre en el mundo real.

Vamos ahora a dar una serie de ejemplos en los que nos iremos acercando paulatinamente a los objetos informáticos. Los últimos ejemplos son para aquellos que ya conocen Java y/o C; sin embargo, estos ejemplos que exigen conocimientos informáticos, no son imprescindibles para entender plenamente el concepto de clase y el de objeto.

Observe que aunque los datos y los métodos se han enumerado verticalmente, no existe ninguna correspondencia entre un dato y el método que aparece a su derecha, es una simple enunciación.

Ejemplo 1

Tomemos la clase león de la que hablamos antes y veamos cuales serían algunos de sus datos y de sus métodos.

Datos	Métodos
Color	Desplazarse
Tamaño	Masticar
Peso	Digerir
Uñas retráctiles	Respirar
Colmillos	Secretar hormonas
Cuadrúpedo	Parpadear
etc.	etc.

Ejemplo 2

Nuestros objetos (los informáticos), como hemos comentado antes, se parecen mucho a los del mundo real, al igual que estos, poseen propiedades (datos) y comportamientos (métodos). Cojamos para nuestro ejemplo un cassette. Veamos cómo lo definiríamos al estilo de OOP.

Datos	Métodos
Peso	Rebobinar la cinta
Dimensiones	Avanzar la cinta
Color	Grabar
Potencia	Reproducir
etc.	etc.

Ejemplo 3

Pongamos otro ejemplo algo más próximo a los objetos que se suelen tratar en informática: un recuadro en la pantalla.

El recuadro pertenecería a una clase a la llamaremos **marco**. Veamos sus datos y sus métodos.

Datos	Métodos
Coordenada superior izquierda	Mostrarse
Coordenada inferior derecha	Ocultarse
Tipo de línea	Cambiar de posición
Color de la línea	
Color del relleno	
etc.	etc.

Ejemplo 4

Vayamos con otro ejemplo más. Este es para aquellos que ya tienen conocimientos de informática, y en especial de lenguaje C o Java.

Una clase es un nuevo tipo de dato y un objeto cada una de las asignaciones que hacemos a ese tipo de dato.

```
int nEdad = 30;
```

Hemos creado un objeto que se llama nEdad y pertenece a la clase *integer* (entero).

Para crear un objeto de la clase marco lo haríamos de forma muy parecida: llamando a la función creadora de la clase marco:

```
Marco oCajaMsg := new Marco();
```

No se preocupe si esto no lo ve claro ahora, esto es algo que veremos con detenimiento más adelante.

Para crear un objeto de tipo Marco, previamente hemos tenido que crear este nuevo tipo de dato. Aun cuando ni en Java ni en C un integer sea internamente un objeto, bien podría serlo; de hecho no hay ninguna diferencia conceptual entre la clase integer y la clase Marco. La primera es una de las clases que vienen con el lenguaje y la segunda la creamos nosotros. Pero del mismo modo a como nEdad es una instancia de la clase integer, oCajaMsg es una instancia de la clase Marco (De hecho, en Java existe la clase Integer, que si es una clase "de verdad" (tanto a nivel externo como interno) y de la que se instancias objetos de tipo integer.). Como se ve, básicamente la OOP lo que nos permite es ampliar los tipos de datos con los que vamos a trabajar: esto que dicho así, resulta tan tan simple, si lo piensa un poco, verá que es de una potencia nunca vista antes.

Un objeto lo podemos considerar como un array multidimensional, donde se almacenan los datos de ese objeto (las coordenadas, el color, etc. en el ejemplo del marco) y sus métodos, que no serían más que punteros a funciones que trabajan con esos datos.

De hecho, y para ser más precisos, en el array de un objeto no se guardan los punteros a los métodos de ese objeto, sino los punteros a otros arrays donde se hallan estas funciones, para ahorrar de este modo memoria. También se guardan punteros a las clases superiores si las hubiese para buscar allí los métodos que no encontremos en la clase a la que el objeto pertenece: si un método no es hallado en una clase, se supone que debe pertenecer a la clase padre (clase superior). A esto le llamamos herencia, pero de esto hablaremos extensamente más adelante.

Ejemplo 5

Como hemos dicho, una clase es un nuevo tipo de dato y objetos son cada una de las asignaciones que hacemos a ese tipo de dato.

En C un objeto lo podemos considerar como un *Struct*. Esta estructura albergaría los datos del objeto, y los punteros a las funciones que formarían el conjunto de sus métodos, más otros punteros a las clases superiores (padre).

Cada una de las instanciaciones (asignaciones) de variables que hagamos a un Struct, equivaldrá a crear un nuevo objeto de una clase.

Las clases aparecen en C como Structs muy mejoradas. De hecho, fueron los Structs quienes sugirieron y dieron lugar a las clases y son sus antecesores informáticos.

Para que entendamos hasta donde llega esta similitud, le informaremos que existe un programa que recibe un código fuente hecho en C++ (C con OOP) y devuelve otro código fuente equivalente al anterior, pero en C normal.

Herencia

Esta es la cualidad más importante de un sistema OOP, la que nos dará mayor potencia y productividad, permitiéndonos ahorrar horas y horas de codificación y de depuración de errores. Es por ello que me niego a considerar que un lenguaje es OOP si no incluye herencia, como es el caso de Visual Basic (al menos hasta la versión 5, que es la última que conozco).

Como todos entendemos lo que es la herencia biológica, continuaremos con nuestro ejemplo taxonómico del que hablábamos en el epígrafe anterior.

La clase león, como comentábamos antes, hereda cualidades -métodos, en lenguaje OOP- de todas las clases predecesoras -padres, en OOP- y posee métodos propios, diferentes a los del resto de las clases. Es decir, las clases van especializándose según se avanza en el árbol taxonómico. Cada vez que creamos una clase heredada de otra (la padre) añadimos métodos a la clase padre o modificamos alguno de los métodos de la clase padre.

Veamos qué hereda la clase león de sus clases padre:

Clase	Qué hereda
Vertebrados -->	Espina dorsal
Mamíferos -->	Se alimenta con leche materna
Carnívoros -->	Al ser adultos se alimenta de carne

La clase león hereda todos los métodos de las clases padre y añade métodos nuevos que forman su clase distinguiéndola del resto de las clases: por ejemplo el color de su piel.

Observemos ahora algo crucial que ya apuntábamos antes: dos subclases distintas, que derivan de una misma clase padre común, pueden heredar los métodos de la clase padre tal y como estos han sido definidos en ella, o pueden modificar todos o algunos de estos métodos para adaptarlos a sus necesidades.

En el ejemplo que exponíamos antes, en la clase león la alimentación es carnívora, mientras que en la clase hombre, se ha modificado éste dato, siendo su alimentación omnívora.

Pongamos ahora un ejemplo algo más informático: supongamos que usted ha construido una clase que le permite leer números enteros desde teclado con un formato determinado, calcular su IVA y almacenarlos en un fichero. Si desea poder hacer lo mismo con números reales (para que admitan decimales), solo deberá crear una nueva subclase para que herede de la clase padre todos sus métodos y redefinirá solo el método de lectura de teclado. Esta nueva clase sabe almacenar y mostrar los números con formato porque lo sabe su clase padre.

Las cualidades comunes que comparten distintas clases, pueden y deben agruparse para formar una clase padre -también llamada **superclase**-. Por ejemplo, usted podría **derivar** las clases *presupuesto*, *albarán* y *factura* de la superclase *pedidos*, ya que estas clases comparten características comunes. De este modo, la clase padre poseería los métodos comunes a todas ellas y sólo tendríamos que añadir aquellos métodos propios de cada una de las subclases, pudiendo reutilizar el código escrito en la superclase desde cada una de las clases derivadas. Así, si enseñamos a la clase padre a imprimirse,

cada uno de los objetos de las clases inferiores sabrán automáticamente y sin escribir ni una solo línea más de código imprimirse.

¡Genial! estará pensando usted, desde luego que lo es.

La herencia como puede intuir, es la cualidad más importante de la OOP, ya que le permite reutilizar todo el código escrito para las superclases re-escribiendo solo aquellas diferencias que existan entre éstas y las subclases.

Veamos ahora algunos aspectos más técnicos de la herencia:

A la clase heredada se le llama, *subclase* o *clase hija*, y a la clase de la que se hereda *superclase* o *clase padre*.

Al heredar, la clase heredada toma directamente el comportamiento de su superclase, pero puesto que ésta puede derivar de otra, y esta de otra, etc., una clase toma indirectamente el comportamiento de todas las clases de la rama del árbol de la jerarquía de clases a la que pertenece.

Se heredan los datos y los métodos, por lo tanto, ambos pueden ser redefinidos en las clases hijas, aunque lo más común es redefinir métodos y no datos.

Muchas veces las clases –especialmente aquellas que se encuentran próximas a la raíz en el árbol de la jerarquía de clases– son abstractas, es decir, sólo existen para proporcionar una base para la creación de clases más específicas, y por lo tanto no puede instanciarse de ellas; son las clases virtuales.

Una subclase hereda de su superclase sólo aquellos miembros visibles desde la clase hija y por lo tanto solo puede redefinir estos.

Una subclase tiene forzosamente que redefinir aquellos métodos que han sido definidos como abstractos en la clase padre o padres.

Normalmente, como hemos comentado, se redefinen los métodos, aun cuando a veces se hace necesario redefinir datos de las clases superiores. Al redefinir un método queremos o bien sustituir el funcionamiento del método de la clase padre o bien ampliarlo.

En el primer caso (sustituirlo) no hay ningún problema, ya que a la clase hija la dotamos con un método de igual nombre que el método que queremos redefinir en la clase padre y lo implementamos según las necesidades de la clase hija. De este modo cada vez que se invoque este método de la clase hija se ejecutará su código, y no el código escrito para el método homónimo de la clase padre.

Pero si lo que queremos es ampliar el funcionamiento de un método existente en la clase padre (lo que suele ser lo más habitual), entonces primero tiene que ejecutarse el método de la clase padre, y después el de la clase hija. Pero como los dos métodos tienen el mismo nombre, se hace necesario habilitar alguna forma de distinguir cuando nos estamos refiriendo a un método de la clase hija y cuando al del mismo nombre de la clase padre.

Esto se hace mediante el uso de dos palabras reservadas, las cuales pueden variar dependiendo del lenguaje que se utilice, pero que normalmente son: **this** (en algunos lenguajes se utiliza la palabra reservada *Self*) y **super**:

this

Con esta palabra, podemos referirnos a los miembros de la clase.

De hecho, siempre que dentro del cuerpo de un método nos refiramos a cualquier miembro de la clase, ya sea una variable u otro método, podemos anteponer *this*, auque en caso de no existir duplicidad, el compilador asume que nos referimos a un miembro de la clase.

Algunos programadores prefieren utilizar *this* para dejar claro que se está haciendo referencia a un miembro de la clase.

super

Al contrario que *this*, *super* permite hacer referencia a miembros de la clase padre (o a los ancestros anteriores, que no hayan sido ocultados por la clase padre) que se hayan redefinido en la clase hija.

Si un método de una clase hija redefine un miembro –ya sea variable o método– de su clase padre, es posible hacer referencia al miembro redefinido anteponiendo *super*.

Veamos un ejemplo (suponemos que solo los datos especificados de la clase A son visibles desde la clase B):

Clase A			
Datos	Visible	Métodos	Visible
AD1	No	AM1	No
AD2	Si	AM2	Si
AD3	Si	Am3	Si

La clase B hereda de la clase A:

Clase B			
Datos	Heredado	Métodos	Heredado
AD2	Si	AM2	Si
AD3	Si	AM3	Si
BD1	No	BM1	No
BD2	No	BM2	No

Si en la clase B quisieramos redefinir el método AM2 de la clase A ampliándolo tendríamos que referirnos a él mediante *super* del siguiente modo:

```
public void AM2::B
{
   super.AM2();    // Invocamos la ejecución del método AM2::A
```

```
    ..............  // Resto del la implementación de AM2::B
    ..............
}
```

::B indica que el método AM2 está dentro de la clase B, al igual que AM2::A indica que es método AM2 de la clase A. Las palabras en azul indican el alcance del método y el valor que este devuelve (no se preocupe si no lo entiende ahora). En rojo hemos indicado el centro del tema que estamos tratando.

Veamos otro caso: supongamos que desde el método BD1 (BD1::B) queremos invocar el método AM2 de la clase B (AM2::B) y que desde el método BD2 (BD2::B) queremos invocar el método AM2 de la clase A (AM2::A).

```
public void BD1::B
{
   AM2();       // Invocamos la ejecución del método AM2::B

   this.AM2()   // Lo mismo que la línea anterior, pero usando this
   ............ // Resto del la implementación de BD1::B
   ............
}

public void BD2::B
{
   super.AM2();  // Invocamos la ejecución del método AM2::A

   ............  // Resto del la implementación de BD2::B
   ............
}
```

Encapsulación

Hemos definido antes un objeto como un conjunto de datos y métodos. Dijimos también, que los métodos son procedimientos que trabajan con los datos del objeto. Veamos esto ahora con más detenimiento.

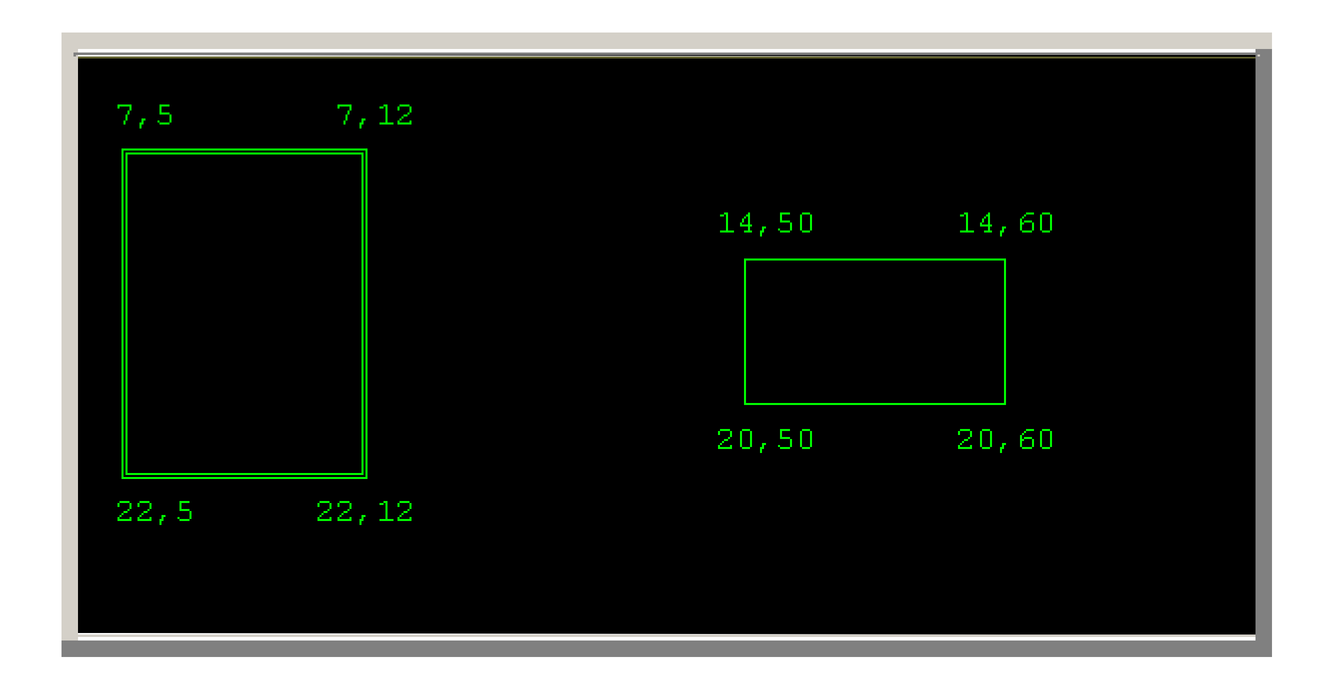
Cuando definíamos el concepto de objeto un poco más arriba, dimos varios ejemplos de objetos; el tercero se refería a un marco (un recuadro) que podía visualizarse en nuestra pantalla de ordenador. La clase marco tenía las siguientes características:

Datos	Métodos
Coordenada superior izquierda	Mostrarse
Coordenada inferior derecha	Ocultarse
Tipo de línea	Cambiar de posición
Color de la línea	
Color del relleno	

Evidentemente, podríamos (y así deberíamos hacerlo) derivar la *clase marco* de la *superclase visual*, pero para simplificar lo más posible y centrarnos en lo que ahora tratamos, vamos a considerar esta clase como totalmente independiente de las demás.

Los datos de la clase son siempre los mismos para todos los objetos de esa clase, e igualmente los métodos, pero para cada instanciación de la clase -cada uno de los objetos pertenecientes a esa clase- los valores de esos datos serán distintos; los métodos trabajarán con cada uno de estos valores de los datos dependiendo del objeto del que se trate.

Veámoslo gráficamente, supongamos que la tabla es nuestro monitor en modo texto 80x24 y los recuadros son dos objetos de la clase marco.



Hemos puesto en cada una de las esquinas las coordenadas de los vértices de los objetos marco. Estas coordenadas son, desde luego, aproximadas y se las hemos escrito en el siguiente formato: primero la ordenada y luego la abcisa (<Ypos>,<Xpos>), situando el origen de coordenadas en el ángulo superior izquierdo de la pantalla con origen en 0,0. De hecho, para definir un cuadrado en un plano cartesiano solo es necesario definir los dos vértices superior izquierdo e inferior derecho.

De este modo, podemos apreciar cómo ambos objetos poseen los mismo datos:

Datos	**Valores**	
	Obj-1	**Obj-2**
Coordenada superior izquierda	7, 5	14,50
Coordenada inferior derecha	22,12	20,60
Color de la línea	Verde	Verde

Tipo de línea del recuadro	Doble	Simple

Los datos son los mismos, pero los valores que toman esos datos son diferentes para cada objeto.

Ahora, podemos aplicar los métodos a los datos. Estos métodos producirán distintos resultados según a qué datos se apliquen. Así, el *objeto marco 1*, al aplicarle el método **Mostrarse**, aparece en la parte izquierda, rectangular verticalmente y con línea doble, mientras que el *objeto marco 2*, al aplicarle el mismo método, aparece en la parte izquierda, cuadrado y con línea simple.

Si quisiéramos ahora aplicarle el método **Cambiar de posición** al *objeto marco 1*, este método debería seguir los siguientes pasos y en este orden.

Llamar al método **Ocultarse** para este objeto.

Cambiar los datos de coordenadas para este objeto.

Llamar al método **Mostrarse** para este objeto.

Vemos así cómo un método de una clase puede llamar a otros métodos de su misma clase y cómo puede cambiar los valores de los datos de su misma clase. De hecho es así como debe hacerse: los datos de una clase sólo deben ser alterados por los métodos de su clase; y no de forma directa (que es como cambiamos los valores de las variables de un programa). Esta es una regla de oro que no debe olvidarse: **todos los datos de una clase son privados y se accede a ellos mediante métodos públicos**.

Veamos cómo se realiza esta acción según los dos modos comentados. Tomemos como ejemplo un objeto perteneciente a la clase *marco*, modificaremos su dato `nY1` (coordenada superior izquierda) de dos modos distintos: directamente y mediante el método `PonerY1()`.

Cambio directo: oCajaGeneral.nY1 = 12;

Cambio mediante invocación de método: `oCajaGeneral.PonerY1( 12 );`

Es más cómodo el primer método, ya que hay que escribir menos para cambiar el valor del dato y además, a la hora de construir la clase, no es necesario crear un método para cambiar cada uno de los datos del objeto. Sin embargo, y como ya hemos comentado, la OOP recomienda efusivamente que se utilice el segundo procedimiento. La razón es bien simple: una clase debe ser una estructura cerrada, no se debe poder acceder a ella si no es a través de los métodos definidos para ella. Si hacemos `nY1` público (para que pueda ser accedido directamente), estamos violando el principio de encapsulación.

Esta forma de trabajo tiene su razón de ser: en algunos casos, pudiera ser que el método que cambia un dato realiza ciertas acciones o comprobaciones que el programador que está usando un objeto creado por otra persona no conoce, con lo que al manipular los datos del objeto directamente, podemos estar provocando un mal funcionamiento del mismo.

Para evitar que se puedan modificar datos que el usuario del objeto no debe tocar, algunos de ellos se hacen de *solo-lectura*, es decir, se puede saber su valor, pero no alterarlo. A estos datos los llamamos *ReadOnly* -LecturaSolo-.

Sin embargo, hay una excepción a esta regla: se pueden hacer públicos todos los datos que la clase no utiliza. Y usted se preguntará, si la clase no los utiliza ¿Para qué los quiere? Hay un caso especial en el que al usuario de la clase se le proporciona una "bolsa" ("cargo" en Clipper, "tag" en Delphi) para que

él almacene allí lo que quiera. De hecho, este recurso no es muy ortodoxo, ya que que lo que la teoría dice es que hay que heredar de la clase y añadir lo que uno necesite, pero es un recurso muy práctico, muy cómodo, y tampoco hay que ser más papistas que el Papa.

Dejemos ahora un poco de lado a los datos para centrarnos en otros aspectos de los métodos.

¿Cómo requerimos la actuación de un método?

Enviando un **Mensaje** al objeto.

Al enviar un mensaje, se ejecuta un método, el cual puede llamar a su vez a otros métodos de su clase o de cualquier otra clase o bien cambiar los valores de los datos de ese objeto. Si el programador tiene que alterar los valores de los datos de un objeto deberá mandar un mensaje a ese objeto; lo mismo sucede cuando un objeto tiene que cambiar los valores de los datos de otro objeto.

Como podemos apreciar, un objeto es como una caja negra, a la que se le envía un mensaje y éste responde ejecutando el método apropiado, el cual producirá las acciones deseadas. un objeto, una vez programado es solo manipulable a través de mensajes.

A este intrínseco vínculo entre datos y métodos y al modo de acceder y modificar sus datos es a lo que llamamos **Encapsulación**. Gracias a la encapsulación, una clase, cuando ha sido programada y probada hasta comprobar que no tiene fallos, podemos usarla sin miedo a que al programar otros objetos estos puedan interferir con los primeros produciendo efectos colaterales indeseables que arruinen nuestro trabajo; esto también nos permite depurar (eliminar errores de programación) con suma facilidad, ya que si un objeto falla, el error solo puede estar en esa clase, y no en ninguna otra. Si usted ha programado con técnicas tradicionales sabrá apreciar lo que esto vale.

Polimorfismo

Por polimorfismo entendemos aquella cualidad que poseen los objetos para responder de distinto modo ante el mismo mensaje.

Pongamos por ejemplo las clases *hombre*, *vaca* y *perro*, si a todos les damos la orden -enviamos el mensaje- **Come**, cada uno de ellos sabe cómo hacerlo y realizará este comportamiento a su modo.

Veamos otro ejemplo algo más ilustrativo. Tomemos las clases *barco*, *avión* y *coche*, todas ellas derivadas de la clase padre *vehículo*; si les enviamos el mensaje **Desplázate**, cada una de ellas sabe cómo hacerlo.

Realmente, y para ser exactos, los mensaje no se envían a las clases, sino a todos o algunos de los objetos instanciados de las clases. Así, por ejemplo, podemos decirle a los objetos *Juan Sebastián el Cano* y *Kontiqui*, de la clase *barco* que se desplacen, con los que el resto de los objetos de esa clase permanecerán inmóviles.

Del mismo modo, si tenemos en pantalla cinco recuadros (marcos) y tres textos, podemos decirle a tres de los recuadros y a dos de los textos que cambien de color y no decírselo a los demás objetos. Todos estos sabrán cómo hacerlo porque hemos redefinido para cada uno de ellos su método **Pintarse** que bien podría estar en la clase padre *Visual* (conjunto de objetos que pueden visualizarse en pantalla).

En programación tradicional, debemos crear un nombre distinto para la acción de pintarse, si se trata de un texto o de un marco; en OOP el mismo nombre nos sirve para todas las clases creadas si así lo queremos, lo que suele ser habitual. El mismo nombre suele usarse para realizar acciones similares en clases diferentes.

Si enviamos el mensaje **Imprímete** a objetos de distintas clases, cada uno se imprimirá como le corresponda, ya que todos saben cómo hacerlo.

El polimorfismo nos facilita el trabajo, ya que gracias a él, el número de nombres de métodos que tenemos que recordar disminuye ostensiblemente.

La mayor ventaja la obtendremos en métodos con igual nombre aplicados a las clases que se encuentran próximas a la raíz del árbol de clases, ya que estos métodos afectarán a todas las clases que de ellas se deriven.

Sobrecarga

La sobrecarga puede ser considerada como un tipo especial de polimorfismo que casi todos los lenguajes de OOP incluyen.

Varios métodos (incluidos los "constructores", de los que se hablará más adelante) pueden tener el mismo nombre siempre y cuando el tipo de parámetros que recibe o el número de ellos sea diferente.

De este modo, por ejemplo la clase `File` puede tener tantos método `Write()` como tipos de datos queramos escribir (no se preocupe si no entiende la nomenclatura, céntrese en la idea):

File::Write(int i);	Escribe un integer
File::Write(long l);	Escribe un long
File::Write(float f);	Escribe un flota
File::Write(string s);	Escribe una cadena
File::Write(string s, boolean b);	Escribe una cadena pasándola a mayúsculas

Existe un tipo especial de sobrecarga llamada sobrecarga de operadores que, de los leguajes OOP conocidos, solo incorpora C++. Es por esto que consideramos que el abordar este tema escapa a la finalidad del presente curso.



Planteamiento de la implementación

De todo lo dicho hasta ahora, usted probablemente habrá adivinado que lo más importante es planificar bien el árbol (jerarquía si lo prefiere) de clases.

Una buena planificación de cada uno de los datos y métodos que debe incluir cada una de las clases, quién deriva de quién y cómo se inter-relacionan es lo más importante para que un sistema de clases funcione correctamente.

A este respecto no hay ninguna regla invariable que seguir, ya que esto es más un arte que una ciencia.

Para que se haga una idea de la importancia que tiene éste diseño, le haremos saber que en los equipos de programadores que trabajan en OOP suele haber varias personas que se dedican exclusivamente al diseño de esta estructura de clases, los datos que cada una de ellas contendrá, los métodos que trabajarán con esos datos y las inter-relaciones de unas clases con otras. Estas personas, suelen ayudarse de herramientas especializadas en facilitar el desarrollo de jerarquías de clases, la más conocida es "Rational Rose". Pero incluso con la ayuda de herramientas de este tipo, la definición de las clases y sus relaciones sigue siendo más un arte que una ciencia. Y en cualquier caso, nunca el modelo diseñado sobre el papel o la herramienta es el modelo finalmente implementado; de hecho, lo más común es definir un modelo inicial e ir re-haciéndolo una y otra vez según se va implementando.

Sin embargo, y pese a todo lo dicho, hay un par de consejos muy generales que pueden ayudar mucho a la hora de crear estos modelos previos a la implementación de las clases. No se si los he leído de otros autores, si son cosecha propia, o una mezcla de ambos (lo más probable); lo que si se, es que el día que me estas ideas dejaron de estar cociéndose en mi subconsciente, y afloraron a mi conciencia como una idea clara y distinta (en palabras de Descartes), me sentí tan contento que me tomé el resto del día libre, para regocijarme en el nuevo hallazgo.

Se perfectamente que la mayor parte de gente cuando estudia un área tecnológica cualquiera, lo que busca son técnicas para aplicar inmediatamente en la resolución de sus problemas; sin embargo, son mucho más valiosos los consejos de personas que llevan muchos trabajando en esas áreas que las técnicas, aun cuando no se sepa habitualmente apreciar su valor.

Una vez dicho esto, voy a dar tres consejos muy simples, que cuando cualquiera entenderá sin el más mínimo esfuerzo, sin embargo, hasta que usted no los asimile y los haga suyos, no podrá comprender el valor que tienen.

Divide y vencerás: Procure construir un método para realizar cada pequeña tarea que necesite. Cuanto menor sea el ámbito de actuación de un método, más probablemente le resultará reutilizable en otro momento y más fácil le resultará codificarlo. Haga lo mismo con las clases: no escatime en su número. Ante la duda, siempre es mejor dividir una clase en dos que agrupar dos en una.

No piense de forma procedural: En OOP no se puede pensar de una forma procedural, una clase no es un conjunto de funciones relacionadas, es un objeto tan real como los demás que están fuera del ordenador. Los objetos son de verdad, no son cajas de almacenamiento de código.

Los métodos no son funciones: No defina métodos como si fueran funciones, sino como acciones inherentes al objeto: así podrá cambiar el funcionamiento interno completamente sin que cambie ni el nombre de la clase ni los nombres de los métodos. Si al cambiar el funcionamiento de una clase el nombre de la clase y de los métodos no tienen sentido es que estaba mal diseñado.

Por otro lado, y para terminar quisiera recomendarle los pasos a seguir para plantear el diseño de las clases:

Identificar el ámbito de trabajo: Para así ver qué clases están en la raiz de la jerarquía (las más abstractas).

Identificar los distintos sub–ámbitos de trabajo.

Especificar los objetos finales: Las relaciones entre las clases del mismo ámbito y las relaciones con las clases de los otros ámbitos.



De la teoría a la realidad

En este capítulo, vamos a darle algunos apuntes sobre cuestiones prácticas relacionadas con la implementación real de un OOPS (Object Oriented Programming System), más que el punto de vista teórico que hemos mantenido hasta ahora a lo largo del presente capítulo. Finalizaremos resumiendo la idea de clase y de objeto.

El operador de envío

Para poder mandar mensajes a los objetos, necesitamos un operador, a este le llamamos el *operador de envío*. Cada lenguaje puede tener el suyo, pero es frecuente que se utilicen los dos puntos ('**:**')o el punto ('**.**') (en C++ es el punto simple, igual que para referirnos a los elementos de los **struct**'s).

Así, si queremos enviarle el mensaje **Caminar** al objeto **Juan** de la clase **hombre**, escribiríamos lo siguiente.

```
hmrJuan.Caminar()
+-++--+¦+-------+
 ¦  ¦  ¦     +---- Mensaje. Invoca el método de igual nombre¹.
 ¦  ¦  +--------- Operador de envío².
 ¦  +------------ Objeto de la clase hombre hmrJuan.
 +--------------- El prefijo hombre denota su clase.
```

1. En algunos lenguajes OOP se puede hacer que un mensaje

```
    invoque un método con nombre distinto.
2. Como hemos comentado, cada lenguaje puede utilizar su propio
   operador de envío.
```

El operador de envío hace que se ejecute la porción del código agrupada bajo el nombre del método, y el método trabajará con los datos propios de la instancia de la clase a ala que se refiera.

Siguiendo el ejemplo de los marcos expuestos en anteriormente, supongamos que tenemos las dos instancias siguiente de la clase Marco (marco1 y marco2):

Datos de la clase Marco	Valores	
	marco1	marco2
Coordenada superior izquierda	7, 5	14,50
Coordenada inferior derecha	22,12	20,60
Color de la línea	Verde	Verde
Tipo de línea del recuadro	Doble	Simple

Cuando se invoca el método Mostrar() de la clase Marco:

```
marco1.Mostrar();

marco2.Mostrar();
```

El método es el mismo, pero cada instancia de la clase tiene unos valores diferentes para sus datos, por lo que el método utilizará estos valores para mostrar los dos objetos, de este modo, el primer marco se mostrará en sus coordenadas (diferentes a las del segundo marco) y con su color propio (igual al de segundo marco) y con su tipo de línea propio (diferente a la del segundo marco).

Como ya comentamos, el objeto es un todo encapsulado, con él viajan sus métodos y sus datos, por lo tanto, al aplicar el mismo método a dos objetos diferentes, se producen resultados diferentes: porque los valores de sus datos son diferentes.

Referencias a sí mismo

Un caso especial ocurre cuando estamos codificando un método de una clase y tenemos que referirnos a un dato o un método del propio objeto. En este caso no podemos enviar un mensaje al objeto, porque aún no hemos instanciado ningún objeto; estamos creando la clase, o por así decirlo, dentro del objeto mismo.

¿Cómo referirnos entonces, desde un objeto al propio objeto? La respuesta, como siempre en OOP, es la misma que en el mundo real: él-mismo, sí-mismo, self (en inglés). Self (Algunos lenguajes utilizan "self", pero lo más común es que utilicen "this": así lo hacen C++ y Java, por ejemplo.) en OOP se

refiere al propio objeto con el que se está trabajando. Por lo tanto, si estamos escribiendo un método de una clase y queremos enviar un mensaje al propio objeto, escribiríamos:

```
this.Ocultar();
```

Para consultar un dato del objeto, actuaremos de igual modo:

```
if( this.nTop < 1 )
{
   .......
   .......
}
```

No olvide que esta referencia mediante *Self* sólo puede hacerla mientras que codifica la clase.

Para más información, refiérase al capítulo donde tratamos la *Herencia*.

Constructores y Destructores

Constructores

Para poder utilizar un objeto, previamente hemos de crearlo, lo que hacemos mediante el *constructor de la clase* (Observe que en virtud a la *sobrecarga* (referida cuando hablamos del *polimorfismo*) puede haber más de un constructor.). Para ello, dependiendo del lenguaje existen dos procedimientos:

Utilizando un método especial, al que se le denota de un modo también especial (normalmente con la palabra reservada "*constructor*"). Este método nos devuelve un objeto nuevo de esa clase. En este caso, a los métodos constructores se les suele llamar *New()*.

Utilizando un operador especial que el lenguaje proporciona y que normalmente se llama "*new*". Es este caso, el constructor o los constructores son notados de una forma especial: en Java, por ejemplo, se notan con el nombre de la clase y no devuelven ningún tipo, ni siquiera "*void*".

Así, para crear un objeto de la clase hombre, llamado Juan, escribiremos lo siguiente:

1. `Hombre hmrJuan = Hombre.New();`
2. `Hombre hmrJuan = new Hombre();`

Le estamos diciendo al método constructor que nos devuelva un nuevo objeto. Supongamos que este objeto tiene tres datos (que para más sencillez son públicos aunque recuerde que nuca debe hacerlo así) y que queremos darle valores a esos datos del objeto: *Edad*, *Estatura* y *Color-de-ojos*. Haremos lo siguiente:

```
hmrJuan.Edad=30;
hmrJuan.Alto=180;
hmrJuan.Ojos=”Marrón”;
```

En este caso, estamos realmente enviando un mensaje al objeto, pero el lugar de acceder a un método, estamos accediendo a un dato. Si usted lo prefiere, puede considerar el operador de envío más el nombre de un dato como el hecho de enviar el mensaje *"CambiarDato()"*.

Habitualmente, los constructores de clase se fabrican de tal modo que podamos hacer las dos cosas a la vez: crear el objeto y dar valores a sus datos, veamos cómo:

```
Hombre hmrJuan = new Hombre( 30, 180, "Marrón" );
Hombre hmrPepe = new Hombre( 12, 145, "Azul" );
Hombre hmrAna  = new Hombre( 24, 175, "Verde" );
```

Normalmente las clases tiene más de un constructor, de esta forma podemos crear objetos e inicializados de distintas formas. Así, podemos tener un constructor de la clase *Hombre* que recibe solo la edad, otro la edad y la estatura, otro la edad, la estatura y el color de ojos, etc. El número y tipo de constructores solo depende de nuestras necesidades y del sentido común.

Existe un constructor especial al que se le llama "*constructor argumento-cero*" y es aquel que no recibe ningún parámetro. Él inicializa el objeto con los valores por defecto. De este modo, podríamos tener un constructor por defecto de la clase *Hombre* que pusiera el dato *edad* a 30, la *estatura* a 175 y el *Color-de-ojos* a marrón, que son los valores más comunes para objetos de esta clase. Así podríamos hacer:

```
Hombre hrmEstandar = new Hombre()
```

Este método internamente se limitaría a hacer lo siguiente:`

```
Hombre::Hombre()
{
   this.edad = 30;
   this.estatura = 175;
   this.ojos = "marrón";
}
```

Es también importante, el concepto de "*constructor por defecto"*. Muchos lenguajes de OOP, permiten definir una clase sin crear un constructor para la clase. El leguaje, entonces, utiliza el constructor por defecto (interno al lenguaje) para crear objetos de esa clase. Este método interno, normalmente se limita a reservar el espacio de memoria necesario para almacenar los datos del objeto, pero estos datos no están inicializados o no lo están correctamente, ya que el constructor por defecto no puede saber qué valores son los apropiados para los datos de la clase.

De hecho, todos nuestros constructores son llamados por el lenguaje después de que se haya invocado el constructor por defecto, ya que éste es realiza las tareas de bajo nivel (reserva de memoria, manejo de la tabla de símbolos, etc.) necesarias para poder empezar a trabajar con un objeto.

En cualquier caso, la misión del constructor es construir adecuadamente el objeto, es decir, cuando el constructor haya terminado su trabajo, el objeto tiene que estar listo para ser usado.

Destructores

Al igual que existen constructores, en la mayoría de lenguajes de OOP, disponemos de *destructores*. Este es método es muy similar en su operatoria al constructor: existe uno interno (destructor por defecto) que siempre es llamado cuando la variable que contiene un objeto sale fuera de ámbito, y que llama, caso de existir al destructor que nosotros hayamos fabricado.

La funcionalidad del destructor por defecto es deshacer todo lo que constructor por defecto realizó: eliminar las referencias en la tabla de símbolos, liberar la memoria ocupada, etc.

En C++, por ejemplo, el destructor tiene el mismo nombre de la cla se, pero con la virgulilla ('~')como prefijo, en Java, sin embargo, siempre se llama *finalize()*.

Accesibilidad de Datos y Métodos

Modificadores de Accesibilidad

Los modificadores de acceso indican la visibilidad que una variable o un método tienen. Tanto bs distintos tipos posibles, como la palabra reservada para denotarlos dependen, como es lógico, de cada lenguaje. De todos modos, todos los lenguajes OOP incluyen al menos los tres siguientes:

Públicos

Son visibles dentro y fuera de la clase sin restricción alguna. La palabra reservada más común para denotarlos es "*public*".

Como ya hemos comentados, los datos no deben ser nunca públicos, ya que romperían el principio de *Encapsulación* que debe seguir todo proyecto OOP.

Protegidos

Estos miembros de la clase (ya sean datos o métodos) son visibles desde dentro del la clase y desde cualquier otra clase heredada, es decir, clases hijas (o subclases, como prefiera llamarlas). La palabra reservada más común para denotarlos es "*protected*" o "*friend*".

Privados

Lo miembros privados son solo accesibles desde dentro de la clase donde existen. La palabra reservada más común para denotarlos es "*private*".

La sintaxis más habitual es la siguiente:

```
[<public|protected|private>] <TipoVariable> <NombreVariable>;
```

Ejemplos:

```
Public String sNombre;
Protected int nEdad;
Private long nAcceso;
```

`[<public|protected|private>] <TipoRetorno> <NombreMétodo> ...`

Ejemplos:

```
Public void Imprimir();
Protected int Calcular();
Private string Grabar();
```

Los datos (variables miembro) y métodos (funciones miembro) de una clase siempre son visibles desde la propia clase, no existe pues –ni tendría sentido– un modificador de acceso que permita acceder a un dato o un método desde fuera de la clase, pero no desde dentro de ella.

Modificadores de Contenido

Para ser correctos, estos modificadores deberían tratarse, como su nombre indica, no el capítulo dedicado a la accesibilidad, sino en algún otro dedicado a los modificadores de contenido. Sin embargo, y puesto que solo vamos a tratar un modificador de este tipo, nos hemos permitido incluirlo en este capítulo.

Es cierto que la mayoría de los lenguajes de OOP incluyen más de un modificador de contenido, tanto para datos como para métodos, pero puesto que casi todos lenguajes coinciden solo en uno, aquí trataremos solamente el modificador de contenido para datos estáticos.

Los modificadores de contenido afectan a cómo va a ser tratado el contenido de la variable. Así, una variable *estática* mantiene su contenido para todas las instancias de la clase que se hagan, así como para las subclases que de ella se hereden.

A estas, se les llama *variables de la clase*, como contraposición a las *variables de instancia*. Mientras que las variables de instancia se inicializan para cada nueva instancia que se haga de la clase, es decir, existe una copia por cada instancia de la clase, de las variables de la clase existe una sola instancia, independientemente del número de instanciaciones que de la clase se hagan. De este modo, todos los objetos comparten un lugar de almacenamiento común.

El ejemplo más típico de variable de la clase es un contador del número de objetos existentes de la clase. Para ello, sólo hay que incrementar el contador desde el constructor de la clase y decrementarlo desde el destructor. Véase *Constructores y destructores*.

El código –muy simplificado– podría ser algo así:

```
class Cuenteo
{
   static long nObj = 0;
   // Constructor
   public Cuenteo()
   {
      nObj++;
   }
   // Destructor
   protected void finalize()
   {
      nObj--;
   }
   // Obtener el número de objetos existentes
   public long GetObj()
   {
      return nObj;
   }
}
```

Interfaz frente a implementación

El tema de la accesibilidad no lleva indefectiblemente a otro tema, más teórico, con el que está íntimamente relacionado: qué debe ser público y qué protegido y qué privado.

Se le llama "interfaz de la clase" a los métodos públicos de la misma (no nos cansaremos de repetir que los datos no deben ser nunca públicos), e "implementación de la clase" al resto de métodos y a todos los datos.

Si lo piensa bien, el nombre de "interfaz" tiene sentido, ya que supone el conjunto de métodos que nos permiten interactuar con la clase.

Como los datos no pueden ser públicos (y raramente son protegidos) debe haber una parte de la interfaz que proporcione acceso a los mismos. En este sentido, en los últimos años han aparecido dos tendencias bien diferentes:

Permitir el uso de métodos con sintaxis de datos

Muchos lenguajes OOP, como es el caso de Delphi, por ejemplo, permiten indicar que a un determinado dato se accede mediante un método. Normalmente estos lenguajes permiten definir un método para leer el valor del dato (get) y otro método para establecer el valor del dato (set). Esto permite que al utilizar una instancia de la clase parezca que se está accediendo a un dato de la calse de forma directa, como si el dato fuera público, cuando realmente se accede a él mediante un método. Una vez definida la clase, es el motor OOP del lenguaje el que hace todas las conversiones necesarias.

Por ejemplo, teniendo la clase Caja, y suponiendo que para leer el valor del dato nAltura se ha definido el método getAltura() y para modificar el valor del dato se ha definido el método setAltura(); las siguientes instrucciones serían convertidas automáticamente por el lenguaje:

Código	**Transformación**
oCaja1.nAltura + oCaja2.Altura	oCaja1.getAltura() + oCaja2.getAltura()
oCaja1.nAltura = 15	oCaja.setAltura(15)

Utilizar una nomenclatura especial para este tipo de métodos. Este sistema es el que utilizan lenguajes como Java, el cual asume que todos los métodos que tienen como sufijo *get<nombre>()* son para recoger el valor de un dato y los *set<nombre>()* son para asignar el valor a un dato.

Clase y Objeto

Resumamos brevemente las ideas más importante:

Una **clase** es un conjunto de reglas de creación y comportamiento de los objetos.

Un **objeto** es un conjunto de datos que se comporta de acuerdo a las reglas de su clase.

Resumamos para terminar qué es, internamente, una clase y qué un objeto.

Una clase es un conjunto de funciones -métodos- e información para construir objetos de la clase -los datos-. Los métodos están almacenados en la clase y trabajan con los datos de cada objeto.

Si los métodos se guardan fuera del objeto es solo para ahorrar memoria, ya que así evitamos incluir los métodos en cada uno de los objetos que fabriquemos, o dicho en argot de OOP, en cada una de las instanciaciones que de la clase hagamos.

La clase contiene los métodos y los objetos los datos.

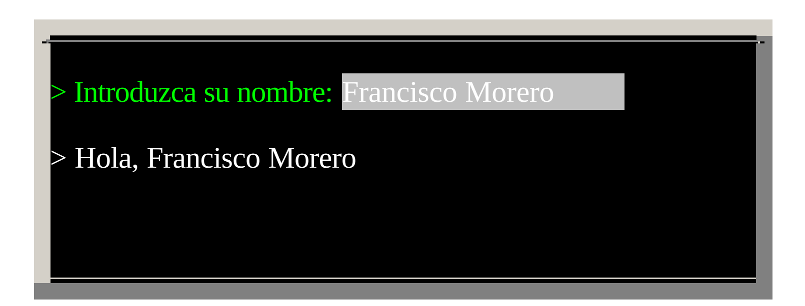
La clase también tiene que ser capaz de crear objetos, por lo que los datos de la clase (comunes a todos los objetos) tienen que estar declarados en la clase, pero ella solo los usa para generar objetos.

Cuando el constructor de una clase crea un nuevo objeto, lo que está fabricando es una estructura de datos (podemos imaginarlo como un **struct** en C o como un **array** en otros lenguajes), que puede contener valores. Existirán tantos casilleros para almacenar valores como datos existan en la clase. Los constructores pueden inicializar todos o algunos de estos datos si lo deseamos.



Limitaciones e inconvenientes de la OOP

La OOP, como todo en este mundo, también tiene sus limitaciones e inconvenientes. De las primeras, no cabe ni hablar, porque, aun cuando sea el sistema más apropiado del que disponemos para programar, todavía los programadores tenemos que dejarnos los sesos para hacer que una máquina boba diga:



Mucha fuerza bruta, pero poca inteligencia.

De los inconvenientes vamos a tratar ahora. Como ya comentábamos antes, una aplicación realizada desde la perspectiva de la OOP, conlleva un análisis mucho más riguroso y tedioso. Sin embargo, que lo vemos como un inconveniente, también, una vez realizado nos permitirá implementar nuestra aplicación en un tiempo mucho menor.

El mayor inconveniente real proviene de un "error" de planteamiento; y como casi siempre, estos son de mucha mayor dificultad a la hora de solucionarlos. El problema, según comenta uno de los mejores analistas de hoy en día, Jeff Dunteman, en un artículo de la revista Dr. Dobbs, es que "La encapsulación y la herencia se hallan en esquinas opuestas de la casa".

Es decir, la encapsulación choca frontalmente con la herencia, y sin embargo, son dos piedras angulares de la OOP.

Por un lado decimos que los objetos deben ser totalmente independientes y autónomos, y por otro, al heredar unas clases de otras, estamos dejando fuera de un objeto perteneciente a una clase hija gran parte de la información que éste necesita para poder comportarse.

A juicio del autor, existe otro problema más inmediato, más real y menos teórico: el que estriba en la imposibilidad de utilización conjunta de objetos de distintos programadores.

Veamos un ejemplo simple, supongamos que estoy montándome un coche en mi garaje, puedo comprar un carburador Fiat y montarlo en un coche Opel, unos asientos Volvo y montarlo en una carrocería Citroën; esto mismo no puedo hacerlo en OOP.

Si Microsoft fabrica la clase *Menu*, que deriva de la clase *Visual* y Borland fabrica la clase *Ventana* aunque también derive de la clase *Visual*, yo no puedo coger el menú de una y la ventana de la otra, a menos que todas las clases superiores (en este caso solo una) sean exactamente iguales: tengan el mismo nombre, contengan los mismos datos y los mismos métodos y en estos, todos los parámetros deben coincidir en orden y en tipo.

Este problema por ahora no tiene solución, y lo peor es que no se vislumbra que la tenga en un futuro próximo, ya que para ello sería necesario normalizar las clases (al menos las más habituales), pero si no nos ponemos de acuerdo para utilizar un mismo HTML ¿Cómo nos vamos a poner de acuerdo para esto?



Glosario

Aquí tiene algunos de los términos más comúnmente utilizados en OOP, el autor reconoce que faltan muchos, pero hacer una recopilación exhaustiva de los mimos es algo que el autor reconoce como una tarea que escapa a sus posibilidades y a su paciencia.

A

Árbol de jerarquía

La estructura de un jerarquía de clases se suele representar como un árbol. En la raíz de este árbol se halla la superclase, y de ella se van ramificando todas las subclases, conectándose con ramas aquellas que derivan de otras. Una clase derivada puede estar conectada con más de una clase padre si existe herencia múltiple para esa clase hija.

C

Clase

Conjunto de datos y métodos.

Conjunto de reglas de creación y comportamiento de los objetos.

Clase abstracta

Clase de apoyo, que construimos solo para derivar de ella otras clases, pero de la que no se puede hacer ninguna instanciación. También

llamada *Clase Virtual*.

Clase base Llamamos así a la clase que se halla al inicio del árbol de las jerarquías de clases. La raíz de ese árbol es la clase base o superclase.

Clase compuesta Véase bajo Clase contenedora.

Clase contenedora Al hecho de crear nuevas clases utilizando otras clases como componentes, se le llama *composición*, y a la clase compuesta se le llama *contenedora*.

Es decir, si alguno de los datos de una clase es a su vez una instancia de otra clase (o de sí misma) a la clase se le llama *contenedoras* y a las clases albergadas *contenidas*.

Clase derivada Una clase deriva de otra cuando hemos aplicado la herencia de una sobre otra. La clase B deriva de la clase A cuando B hereda los datos y métodos de A.

Clase hija Clase que es derivada directamente de otra. Decimos que la clase B es hija de la clase A si B deriva directamente de A (está conectada directamente en el árbol de jerarquías de las clases).

Clase padre La clase de la cual otra deriva directamente. Decimos que la clase A es padre de la clase B si B deriva directamente de A (está conectada directamente en el árbol de jerarquías de las clases).

Clase virtual Véase bajo Clase abstracta.

Constructor Método que devuelve un objeto de la clase a la que el constructor pertenece. Los constructores, habitualmente permiten asignar valores a todos o algunos datos del objeto que van a generar.

Constructor por defecto En muchos lenguaje que implementan OOP, se permite definir una clase sin crear un constructor para la clase. El leguaje, entonces, utiliza el constructor por defecto para crear objetos de esa clase. Este método, devuelve un objeto de su clase bien sin valor alguno o con los valores por defecto mínimos para que el objeto pueda ser inicializado.

Constructor cero-argumento Aquél constructor que no recibe argumentos, y que por lo tanto es el encargo de inicializar el objeto con todos sus valores por defecto.

D

Dato Variable perteneciente a una clase. El conjunto de variables de una clase forman los datos de esa clase.

Aunque depende del leguaje suelen darse al menos los siguientes

	tipos en cuanto a la accesibilidad: Público: Cualquier método puede acceder a ellos. Privado: Solo pueden ser utilizados por funciones miembro. Protegido: Accesibles desde la clase y las clases heredadas de ella.
Destructor	Método que libera cualquier recurso requerido por el objeto durante su creación o existencia.
Destructor por defecto	Al igual que existe un constructor por defecto, existe un destructor por defecto. Este método, elimina de memoria el objeto al terminarse el ámbito de la variable que lo contiene, recuperando para su uso la porción de memoria que el objeto ocupaba.

E

Early Binding	Ligadura Estática. Véase bajo *Late Binding*.
Encapsulación	Vínculo que existe entre los datos de un objeto y los métodos que los utilizan.

F

Función amiga	Una función F(), perteneciente a la clase A, se dice que es amiga de la clase B, cuando tiene acceso además de a los datos de su clase, a los datos protegidos de la clase B.
Función miembro	Veáse bajo *Método*.

H

Herencia	Capacidad que tienen las clases derivadas o hijas para utilizar toda la información (datos y métodos) de todas las clases superiores.
Herencia múltiple	Cuando una clase hija deriva de más de una clase padre; con lo que adquiere los datos y métodos de todas las clases de las cuales deriva.

I

Instanciar	Genéricamente, crear una nueva variable. En OOP, utilizamos este término para referirnos a la acción de crear un nuevo objeto. Este objeto contendrá espacio para almacenar valores para todos los datos de la clase a la que pertenece, y las referencias a los métodos de su clase -funciones miembro-. En realidad, no contiene en sí los

métodos, que se quedan en la clase. La finalidad que se persigue dejando los métodos en la clase, en lugar de incorporarlos en cada uno de los objetos que instanciemos, es la de ahorrar memoria no duplicando innecesariamente esta información.

L

Late binding

En castellano podríamos traducirlo como *Ligadura Dinámica*.

Habitualmente, es el enlazador el que se encarga de resolver las llamadas a las funciones de nuestro programa, esta resolución de llamadas se hace en tiempo de compilación (de enlazado, para ser exactos), a esto se le llama *Ligadura Estática*.

Pero en OOP, a menudo, no se puede saber qué función es la que va a ser llamada, por lo que esta resolución de llamadas debe hacerse en tiempo de ejecución. A esta resolución "tardía" es a lo que llamamos *Ligadura Dinámica* o *Late Binding*.

M

Método

Se les llama así a las funciones que realizan operaciones con los datos de un objeto. También llamados *funciones miembro*.

Los datos privados de una clase sólo pueden ser modificados por medio de las funciones miembro -todos los métodos de esa clase y sólo los de esa clase-.

Mensaje

El modo de solicitar que un objeto realice una acción, es enviándole un mensaje. Un mensaje normalmente activa cualquiera de los métodos de la clase a la que pertenece el objeto. Aunque hay mensajes que no activan ningún método, como son los de inspección y/o modificación de un dato del objeto. Para enviar un mensaje a un objeto se utiliza el operador de envío (normalmente '.' o ':', dependiendo del lenguaje).

Por ejemplo:

```
btnCancel.Display() // enviar el mensaje de Display()
? btnCancel.nTop    // consultar el dato nTop
```

Miembros

Este concepto agrupa al de *Función Miembro* (el construtor y el destrutor son habitualmente considerados como un tipo especial de funciones miembro) y el de *Datos Miembros* de la clase.



O

Objeto

Cualquiera de la instanciaciones que hacemos de una clase.

Una clase podemos considerarla como un conjunto de datos y

métodos, es decir, como un nuevo tipo de dato (como lo son los tipos carácter, lógico, array, ...).

Un objeto sería una variable cuyo tipo es el la clase a la que pertenece.

OOPS

Object Oriented Programming System (Sistema de Programación Orientado al Objeto). Puede encontrarlo también como "Sistema de Programación Orientado a Objetos". Son dos traducciones distintas de la misma expresión inglesa.

OOUI

Object Oriented User Interface (Interface de Usuario Orientado al Objeto). La OOP suele tener como aplicación m s inmediata, el desarrollo de interfaces de usuario, a estos se les llama OOUI; ya que han sido construidos completamente con objetos.

P

Polimorfismo

Capacidad que tienen los objetos de clases diferentes para actuar frente al mismo mensaje de formas distintas.

R

Receptor

Llamamos así al objeto que recibe un mensaje.

Cuando se implementa una clase, dentro de cada uno de los m,todos de la clase, la variable *this* (en otros lenguajes *Self*) contiene el receptor del mensaje.

Reusabilidad

Denota la permisividad de un sistema de programación para poder utilizar partes de código ya escritas en problemas diferentes.

La OOP es el sistema de programación que proporciona un mayor grado de reusabilidad del código.

S

Self

Véase bajo *Receptor*.

Sobrecarga

Algunos lenguajes de OOP (por ejemplo C++) permiten realizar lo que llamamos sobrecarga de operadores y de funciones.

Por sobrecarga de operadores entendemos la posibilidad de definir para un operador existente comportamientos distintos dependiendo del tipo de los datos con los que trabaje el operador.

Por sobrecarga de funciones entendemos la posibilidad de definir distintas funciones con el mismo nombre, pero con diferente conjunto de argumentos. El lenguaje activar aquella que corresponda al

conjunto de argumentos pasados a la función en cada ocasión. Sin sobrecarga de funciones no puede haber *Polimorfismo*.

Superclase

Cualquier clase de la que derivan una o más clases.

Normalmente, a la clase que se halla directamente por encima de otra determinada, la llamamos clase padre, y aquella de la que derivan todas -la que se halla a la raíz del árbol de jerarquías- la llamamos *Superclase* o *Clase Base*.

Subclase

Cualquier clase que es derivada de otra (u otras si el sistema permite heremcia múltiple) es una subclase.

También llamada Clase Hija o Clase derivada.

T

This

Véase bajo *Receptor*.

V

Variable de clase

Son aquellas de las que solo existe una copia para todas las instancias de la clase. Es decir, todas las instancias de la clase comparten la misma variable en lugar de una copia de la variable que es lo normal.

Virtual, método

Aquel que se define como vacío en una clase superior para que cada subclase lo redefina según sus necesidades.


GRUPO
EIDOS